



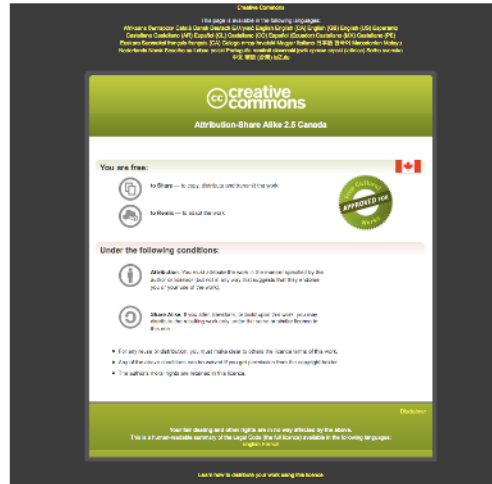
Canadian Bioinformatics Workshops

www.bioinformatics.ca
bioinformaticsdotca.github.io

Supported by

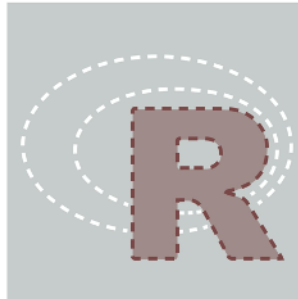


bioinformatics.ca



R Shiny

Gabi Morgenshtern
Introduction to R
June 9-10, 2020



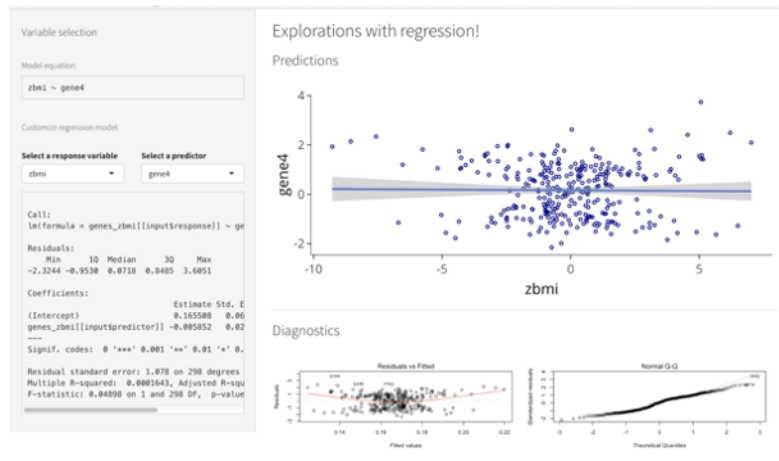
Learning Objectives

- By the end of this lecture, you will:
 - Understand R Shiny's application structure
 - Understand reactivity in Shiny
 - Be able to leverage reactivity in exploratory data analysis
 - Know how to restructure your R scripts into interactive applications
 - Feel empowered to develop and share web applications in R

Shiny: what is it? Why learn it?

- Shiny is a web development framework
 - **Framework:** a software that provides highly customizable functionality to a generalized tool (like a website)
- Shiny is an R package that transforms your R code into an interactive web tool
- Shiny is a great way to share your analysis with your colleagues

What we're building today



5

bioinformatics.ca

While this is a locally hosted website, there are additional tools that would also allow us to host it on a public webpage.

Look at more examples of what's possible here: <https://shiny.rstudio.com/gallery/>

1. Directory structure



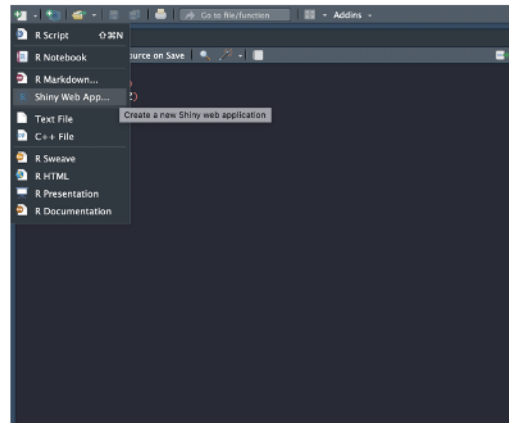
yes



no

Create
workshop_example:
a new folder in your
working directory

Use the R Studio
**Create Shiny Web
App** button to generate
the directory for you



6

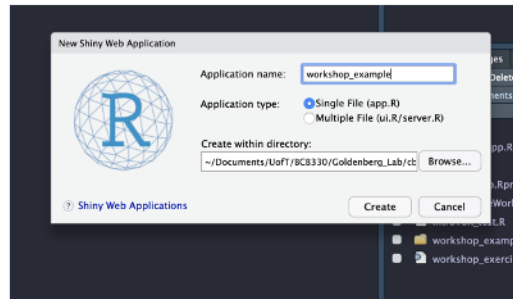
bioinformatics.ca

1. Directory structure



Create
workshop_example:
a new folder in your
working directory

Use the R Studio
Create Shiny Web
App button to generate
the directory for you



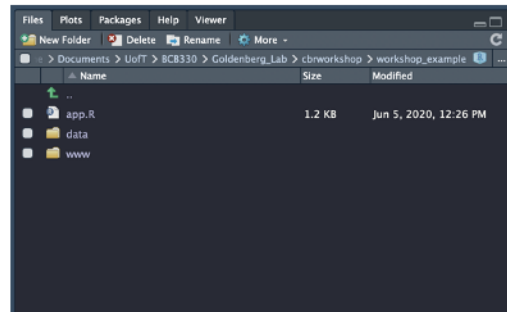
1. Directory structure



Within
workshop_example:

Create new folder **data**

Create new folder **www**



8

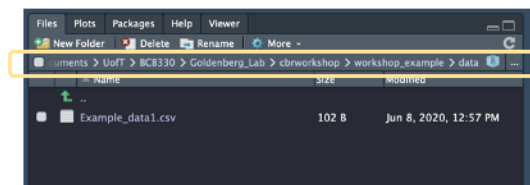
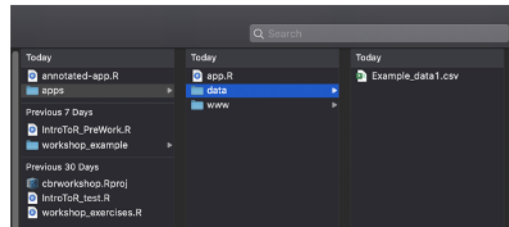
bioinformatics.ca

Your data objects, like RDS files, will be saved in the data directory. Any images you might want to include would be saved in your www directory. When you click the “Run App” button, R will take a look at your application structure and use only these folders to load your Shiny app

1. Directory structure



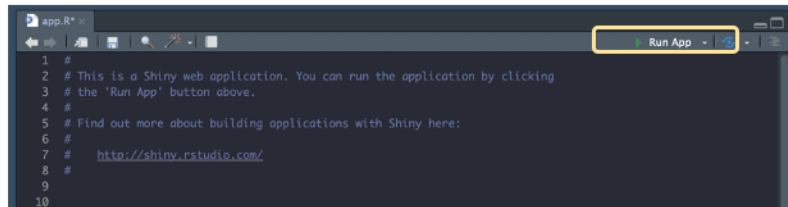
Copy genes_zbmi.rds
into
workshop_example/data/



9

bioinformatics.ca

2. Running the app locally



2. Running the app locally



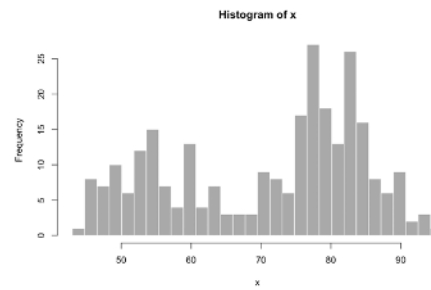
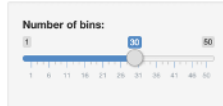
yes



no

~/Documents/UofT/BCB330/Goldenberg_Lab/cbrworkshop/workshop_example - Shiny
http://127.0.0.1:3733 Open in Browser Publish

Old Faithful Geyser Data



11

bioinformatics.ca

3. Anatomy of a Shiny app

The core components of a Shiny **app.R** file:

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

shinyApp(ui = ui, server = server)
```

12

bioinformatics.ca

Though our example app.R has quite a bit more content than this, the text above represents a fully valid and executable app.R file (though this Shiny app won't have any visual or data components to generate)

3. Anatomy of a Shiny app - basics

The core components of a Shiny **app.R** file:

```
library(shiny)  Loading the library that  
                contains the Shiny framework  
  
ui <- fluidPage()  
  
server <- function(input, output) {}  
  
shinyApp(ui = ui, server = server)
```

13

bioinformatics.ca

Though our example app.R has quite a bit more content than this, the text above represents a fully valid and executable app.R file (though this Shiny app won't have any visual or data components to generate)

3. Anatomy of a Shiny app - basics

The core components of a Shiny **app.R** file:

```
library(shiny)
```

```
ui <- fluidPage()
```

Defining all the front-end
(visual) components

```
server <- function(input, output) {}
```

```
shinyApp(ui = ui, server = server)
```

14

bioinformatics.ca

Though our example app.R has quite a bit more content than this, the text above represents a fully valid and executable app.R file (though this Shiny app won't have any visual or data components to generate)

3. Anatomy of a Shiny app - basics

The core components of a Shiny **app.R** file:

```
library(shiny)

ui <- fluidPage()      Defining the back-end
                        (server logic) function

server <- function(input, output) {}

shinyApp(ui = ui, server = server)
```

15

bioinformatics.ca

Though our example app.R has quite a bit more content than this, the text above represents a fully valid and executable app.R file (though this Shiny app won't have any visual or data components to generate)

Note that, while **ui** is an R *object*, the **server** line actually defines a *function that interacts with the ui object*

3. Anatomy of a Shiny app - basics

The core components of a Shiny **app.R** file:

```
library(shiny)

ui <- fluidPage()

server <- function(input, output) {}

shinyApp(ui = ui, server = server)
```

Running the
application

16

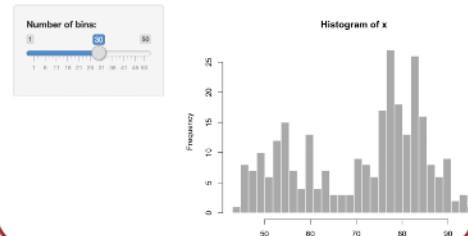
bioinformatics.ca

Though our example app.R has quite a bit more content than this, the text above represents a fully valid and executable app.R file (though this Shiny app won't have any visual or data components to generate)

3. Anatomy of a Shiny app - client side

```
11 # Define UI for application that draws a histogram
12 ui <- fluidPage(
13   # Application title
14   titlePanel("Old Faithful Geyser Data"),
15   # Sidebar with a slider input for number of bins
16   sidebarLayout(
17     sidebarPanel(
18       sliderInput("bins",
19         "Number of bins:",
20         min = 1,
21         max = 50,
22         value = 30)
23     ),
24     # Show a plot of the generated distribution
25     mainPanel(
26       plotOutput("distPlot")
27     )
28   )
29 )
```

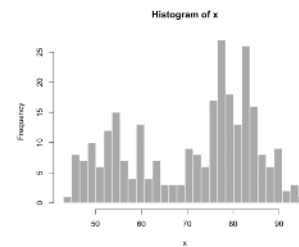
Old Faithful Geyser Data



3. Anatomy of a Shiny app - client side

```
11 # Define UI for application that draws a histogram
12 ui <- fluidPage(
13
14   # Application title
15   titlePanel("Old Faithful Geyser Data"),
16
17   # Sidebar with a slider input for number of bins
18   sidebarLayout(
19     sidebarPanel(
20       sliderInput("bins",
21                 "Number of bins:",
22                 min = 1,
23                 max = 50,
24                 value = 30)
25     ),
26
27     # Show a plot of the generated distribution
28     mainPanel(
29       plotOutput("distPlot")
30     )
31   )
32 )
```

Old Faithful Geyser Data



18

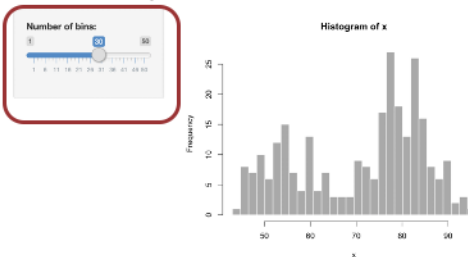
bioinformatics.ca

A Shiny application is built from a set of components available ready-made in the library. Each **xOutput** call corresponds to a **renderX** call in your `server()` function. The full set of the available render/output pairings can be found by referencing <https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

3. Anatomy of a Shiny app - client side

```
11 # Define UI for application that draws a histogram
12 ui <- fluidPage(
13   # Application title
14   titlePanel("Old Faithful Geyser Data"),
15   # Sidebar with a slider input for number of bins
16   sidebarLayout(
17     sidebarPanel(
18       sliderInput("bins",
19         "Number of bins:",
20         min = 1,
21         max = 50,
22         value = 30)
23     ),
24     # Show a plot of the generated distribution
25     mainPanel(
26       plotOutput("distPlot")
27     )
28   )
29 )
```

Old Faithful Geyser Data



19

bioinformatics.ca

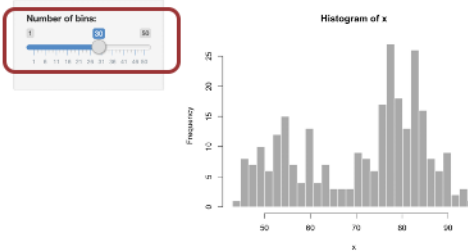
A Shiny application is built from a set of components available ready-made in the library. Each **xOutput** call corresponds to a **renderX** call in your `server()` function. The easiest way to create a screen-size-sensitive, polished application is by using Shiny's preset **xLayout** calls. Here, we see an example of the **sidebarLayout**. Different layouts can be combined essentially at-will to create more complex web pages. A summary of the available layout components and their usage can be found by referencing <https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

<https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

3. Anatomy of a Shiny app - client side

```
11 # Define UI for application that draws a histogram
12 ui <- fluidPage(
13   # Application title
14   titlePanel("Old Faithful Geyser Data"),
15   # Sidebar with a slider input for number of bins
16   sidebarLayout(
17     sidebarPanel(
18       sliderInput("bins",
19         "Number of bins:",
20         min = 1,
21         max = 50,
22         value = 30)
23     ),
24     # Show a plot of the generated distribution
25     mainPanel(
26       plotOutput("distPlot")
27     )
28   )
29 )
```

Old Faithful Geyser Data



20

bioinformatics.ca

Note here the **sliderInput** call. Because the function name includes “**input**”, we know that user interaction here can be passed to the **server()** call, which lets us create an interactive experience in the application using the concept of **reactivity**. Function calls that include “**output**”, on the other hand, mean that they’re displaying an object that was sent from the server function

The easiest way to create a screen-size-sensitive, polished application is by using Shiny’s preset **xLayout** calls. Here, we see an example of the **sidebarLayout**. Different layouts can be combined essentially at-will to create more complex web pages. A summary of the available layout components and their usage can be found by referencing <https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

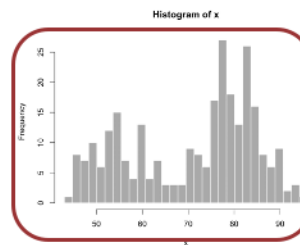
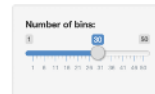
3. Anatomy of a Shiny app - server side

```
# Define server logic required to draw a histogram
server <- function(input, output) {

  output$distPlot <- renderPlot({
    # generate bins based on input$bins from ui.R
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
}
```

Old Faithful Geyser Data



21

bioinformatics.ca

Note here the **renderPlot** call. Because the function name includes “**render**”, we know that this prepares the R expressions within the curly brackets into a plot object to be rendered on the client side (the front end). This **render** call’s return value is assigned to **output\$distPlot**, which can then be read by the **ui** object we defined above (the client-side/front-end/user interface)

There are many different kinds of **renderX** functions, each responsible for a different front-end element’s preparation (ex. strings, general function outputs, data tables). A summary of their functionality is available at <https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>, but the best way to sort out how to use them and what their corresponding **ui xOutput** call would be is by using the **?** operator in RStudio

<https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

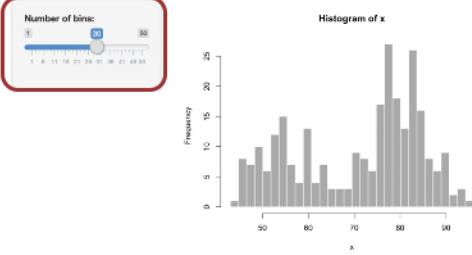
3. Anatomy of a Shiny app - server side

```
# Define server logic required to draw a histogram
server <- function(input, output) {

  output$distPlot <- renderPlot({
    # generate bins based on input$bins from ui.R
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
}
```

Old Faithful Geyser Data



22
bioinformatics.ca

Note here the **input\$bins** call. This is where the concept of **reactivity** comes into play. **server()**, the a function we're defining, has 2 fields, **input** and **output**. User interaction on the front-end is detected by accessing field in the **input** through the object labels we include in the **ui** definition. Information gets back to the user interface to be rendered by creating fields in the **output**. This is the basic principle behind **reactivity**, the theory that drives Shiny's functionality.

Fields in objects are accessed just like columns in dataframes, through the **\$** operator.

There are many different kinds of **renderX** functions, each responsible for a different front-end element's preparation (ex. strings, general function outputs, data tables). A summary of their functionality is available at <https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>, but the best way to sort out how to use them and what their corresponding **ui xOutput** call would be is by using the **?** operator in RStudio

<https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

3. Anatomy of a Shiny app - server side

```

# Define server logic required to draw a histogram
server <- function(input, output) {

  output$histPlot <- renderPlot({
    # generate bins based on input$bins from ui.R
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })

  # Define UI for application that draws a histogram
  ui <- fluidPage(
    # Application title
    titlePanel("Old Faithful Geyser Data"),
    # Sidebar with a slider input for number of bins
    sidebarLayout(
      sidebarPanel(
        sliderInput("bins",
                    "Number of bins:",
                    min = 1,
                    max = 50,
                    value = 30)
      ),
      mainPanel(
        output$histPlot
      )
    )
  )
}

```

Old Faithful Geyser Data

23
bioinformatics.ca

Note here the **input\$bins** call and the defined label **bins**. This is how we create new fields in the **input** that is passed to our **server()** function in the **runApp()** call at the bottom of the Shiny script.

Fields in objects are accessed just like columns in dataframes, through the **\$** operator.

There are many different kinds of **renderX** functions, each responsible for a different front-end element's preparation (ex. strings, general function outputs, data tables). A summary of their functionality is available at <https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>, but the best way to sort out how to use them and what their corresponding **ui xOutput** call would be is by using the **?** operator in RStudio

<https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>

3. Anatomy of a Shiny app - server side

```

# Define server logic required to draw a histogram
server <- function(input, output) {
  output$distPlot <- renderPlot({
    # generate bins based on input$bins from ui.R
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })

  sidebarPanel(
    sliderInput("bins",
      "Number of bins:",
      min = 1,
      max = 50,
      value = 30)
  ),
  # Show a plot of the generated distribution
  mainPanel(
    plotOutput("distPlot")
  )
}

```

Old Faithful Geyser Data

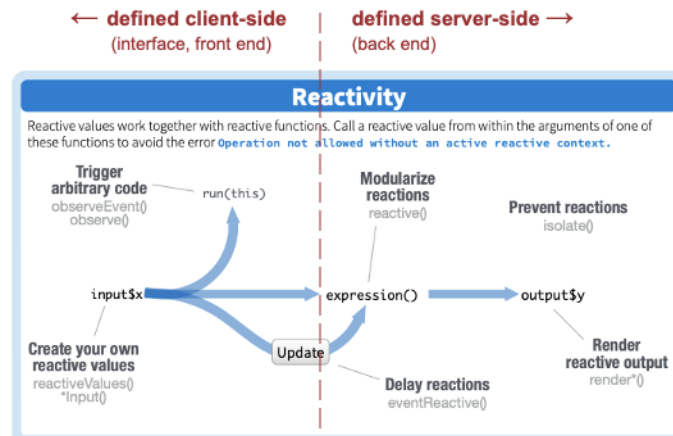
24
bioinformatics.ca

Note here the **output\$distPlot** call and the defined label **distPlot** in the **plotOutput()** call. This is how we create new fields in the **output** that our **ui** object uses to output results from the R expressions we write in the **renderX()** calls in the **server()** function.

Fields in objects are accessed just like columns in dataframes, through the **\$** operator.

There are many different kinds of **renderX** functions, each responsible for a different front-end element's preparation (ex. strings, general function outputs, data tables). A summary of their functionality is available at <https://shiny.rstudio.com/images/shiny-cheatsheet.pdf>, but the best way to sort out how to use them and what their corresponding **ui xOutput** call would be is by using the **?** operator in RStudio

4. How does Shiny work?



25

bioinformatics.ca

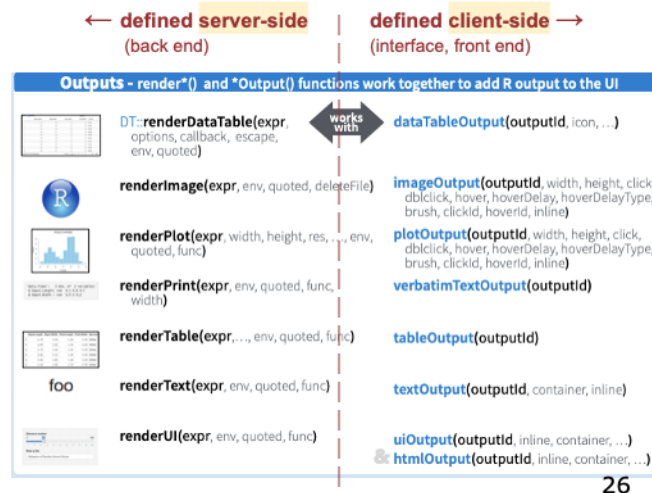
To build our app today, you do not at all need to be an expert on reactivity. You just need to understand the relationship between the interface, input objects, output objects, and the server.

The server function will recognize changes in the **input** object passed to it, and then execute various expressions in a **reactive context**. These expressions include calculations, preparations of plots, modeling functions, etc. They must be called in a function that is suited to accepting reacting values, such as the **renderX()** functions.

Each output in the user interface has a partner on the server-side to re-render its visuals according to changes resulting from user interaction with the application

More details: <https://shiny.rstudio.com/articles/understanding-reactivity.html>

4. How does Shiny work?



26

bioinformatics.ca

To build our app today, you do not at all need to be an expert on reactivity. You just need to understand the relationship between the interface, input objects, output objects, and the server.

The server function will recognize changes in the **input** object passed to it, and then execute various expressions in a **reactive context**. These expressions include calculations, preparations of plots, modeling functions, etc. They must be called in a function that is suited to accepting reacting values, such as the **renderX()** functions.

Each output in the user interface has a partner on the server-side to re-render its visuals according to changes resulting from user interaction with the application

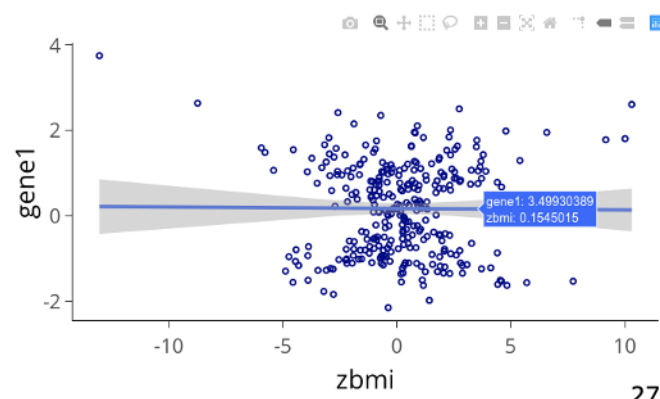
More details: <https://shiny.rstudio.com/articles/understanding-reactivity.html>

5. Making it our own: usability upgrades - `plotly::ggplotly`

Package name

Function name

Predictions



27

bioinformatics.ca

The explicit **package_name::function_name** notation makes your code a lot more readable and maintainable, so it's just generally good coding practice in R. Especially when your scripts become more complex with how they leverage packages. You would still import the package in the same way as always, with **library(package_name)** at the top of the script file

The **plotly** package allows us to introduce interactivity into our Shiny app plots. The **ggplotly** wrapper that we're writing today allows us to very quickly add a ton of useful exploratory functionalities to any **ggplot** object we have in the code

5. Making it our own: usability upgrades - `plotly::ggplotly`

Package name

Function name

Client-side code

```
ui <- fluidPage(  
  ... # whatever other front-end code you're including  
  
  plotlyOutput("lm_plot"), reactive output  
  
  selectizeInput("user_input1", reactive input  
    label = "Select a response variable",  
    choices = colnames(your_dataframe),  
    selected = colnames(your_dataframe)[1],  
    multiple = FALSE,  
    width = '100%'),  
)
```

28

bioinformatics.ca

The explicit **package_name::function_name** notation makes your code a lot more readable and maintainable, so it's just generally good coding practice in R. Especially when your scripts become more complex with how they leverage packages. You would still import the package in the same way as always, with **library(package_name)** at the top of the script file

You can also ask for user input for any part of this function! Fields where user input is accepted would be included elsewhere in your **ui** object. For example, in a dropdown list, as above, or a slider bar, as in the example app we explored.

5. Making it our own: usability upgrades - plotly::ggplotly

Client-side code

```
ui <- fluidPage(  
  ... # whatever other front-end code you're including  
  client-side reference field  
  plotlyOutput("lm_plot"),  
  selectizeInput("user_input1",  
    label = "Select a response variable",  
    choices = colnames(your_dataframe),  
    selected = colnames(your_dataframe)[1],  
    multiple = FALSE,  
    width = '100%'),  
)
```

29

bioinformatics.ca

You can also ask for user input for any part of this function! Fields where user input is accepted would be included elsewhere in your **ui** object. For example, in a dropdown list, as above, or a slider bar, as in the example app we explored.

5. Making it our own: usability upgrades - plotly::ggplotly

Server-side code

```
output$lm_plot <- renderPlotly({  
  
  plotly::ggplotly(  
    ggplot(your_dataframe, aes(x=.data[[input$user_input1]],  
      y=.data[[input$user_input2]])) +  
    geom_point(shape=1, color = "darkblue") +  
    geom_smooth(method=lm) + # Add linear regression line  
    ylab(input$user_input3) +  
    xlab(input$user_input4)  
  )  
})
```

30

bioinformatics.ca

Because our **input\$user_input1** and **input\$user_input2** are strings, we use the double square bracket syntax to access the data that our user is requesting.

NOTE: **.data** refers to the data object you are passing to ggplot in the function

5. Making it our own: usability upgrades - plotly::ggplotly

Server-side code

```
output$lm_plot <- renderPlotly({  
  plotly::ggplotly(  
    ggplot(your_dataframe, aes(x=.data[[input$user_input1]],  
      y=.data[[input$user_input2]])) +  
    geom_point(shape=1, color = "darkblue") +  
    geom_smooth(method=lm) + # Add linear regression line  
    ylab(input$user_input3) +  
    xlab(input$user_input4)  
  )  
})
```

31

bioinformatics.ca

Because our **input\$user_input1** and **input\$user_input2** are strings, we use the double square bracket syntax to access the data that our user is requesting

You can otherwise define your ggplot object in the exact same way as you have up until now

Wrap the entire object in the **plotly::ggplotly()** call to add the extra plotly functionality to it (hover-over data explanations, zoom in/out, etc)

NOTE: **.data** refers to the data object you are passing to ggplot in the function

5. Making it our own: usability upgrades - plotly::ggplotly

Server-side code

client-side reference field

```
output$lm_plot <- renderPlotly({  
  
  plotly::ggplotly(  
    ggplot(your_dataframe, aes(x=.data[[input$user_input1]]  
      y=.data[[input$user_input2]]) +  
    geom_point(shape=1, color = "darkblue") +  
    geom_smooth(method=lm) + # Add linear regression line  
    ylab(input$user_input3) +  
    xlab(input$user_input4)  
  )  
})
```

32

bioinformatics.ca

Because our `input$user_input1` and `input$user_input2` are sent to the server as strings data types, we use the double square bracket syntax to access the data that our user is requesting.

NOTE: `.data` refers to the data object you are passing to ggplot in the function

You can also ask for user input for any part of this function! As long as your ggplot object is wrapper in a **reactive context** like the `renderPlotly` function call, your code can include, and will automatically respond to, any variable value the server recognizes to change on the interface side of the app

We then wrap the entire call of the plot object in a `renderPlotly({})` call, where the **curly brackets** are extremely necessary! They let us write multiple lines of code in this function call at once (which Shiny calls **an expression** in its documentation, and it is the first field to every **renderX** function in the framework)

5. Making it our own: usability upgrades - DT::datatable

Raw data

Show 10 entries Search:

	gene1	gene2	gene3
1	-3.46522442377333	-2.60122519310818	2.48039890636213
2	-0.69170037931616	0.898068229619275	1.15254700905325
3	-0.923830300673967	-0.0761770775059303	-0.876801358112586
4	1.8727264775642	3.08217651143663	-1.79781563035939
5	-1.91574308994114	-0.0943803704528545	-1.84931490088802
6	0.391224719498534	-0.937044770431057	2.03618209777279
7	-1.00342810466625	2.34721938479216	-0.221187099485588
8	4.8223311112455	-3.66624000919505	-0.0792659126123135
9	2.04209276542133	1.6166164428311	0.895363297063054
10	4.1140905464407	-0.459527397133795	4.33980329145326

Showing 1 to 10 of 300 entries

Previous 1 2 3 4 5 ... 30 Next

33

bioinformatics.ca

Tables are often hard to process visually, and coding up your own search bar if not most humans' idea of a fun afternoon. So we use the DT package's wrapper on our dataframe object to quickly introduce interactivity to our datatable front-end component

5. Making it our own: usability upgrades - DT::datatable

Client-side code

```
ui <- fluidPage(  
  ... # whatever other front-end code you're including  
  
  DT::dataTableOutput("lm_df"), reactive output  
  
  ... # whatever other front-end code you're including  
)
```

34

bioinformatics.ca

Given that an **xOutput** function with a reactive output identifier, it must mean we need an **output\$lm_df** component in our server function to create this object. As before, this object will be created in a **renderX()** call, that is also provided by the DT package.

5. Making it our own: usability upgrades - DT::datatable

Server-side code

```
server <- function(input, output) {  
  output$lm_df <- DT::renderDT(your_dataframe) output  
  client-side  
  reference field  
  
  ... # whatever other back-end code you're including  
}
```

35

bioinformatics.ca

Given that an **xOutput** function with a reactive output identifier, it must mean we need an **output\$lm_df** component in our server function to create this object. As before, this object will be created in a **renderX()** call, that is also provided by the DT package.

NOTE: In the **ui** object we are defining the layout and the order in which we call the functions matters. In the server function, on the other hand, the order of the render calls or helper code does not matter, as the entire function will be called again whenever things change on the front-end side

5. Making it our own: layouts

Client-side code

```
ui <- fluidPage(  
  sidebarPanel(  
    # some xOutput() call or other front-end elements  
  ),  
  mainPanel(  
    fluidRow( # total width = 12  
      column(width = 5, offset = 1  
        # some xOutput call or other front-end  
        # elements  
      )  
    )  
  )  
)
```

36

bioinformatics.ca

The user interface layout follows the Bootstrap logic of breaking each visual row into 12 subsections. This allows the browser that compiles all the code we write to resize and redistribute content according to the size of the screen it is displayed on!

Each one of the **fluidRow()** within the **fluidPage()** that defines our website structure can have added width+offset amount of up to 12. The **offset** function field allows us to code space between visual components that coexist in a row

The best way to understand this is by playing around with different layout options, which are quite numerous and include things like navigation bars, tabs, and menus! Find more about how to personalize your Shiny layout here:

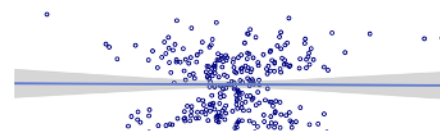
<https://shiny.rstudio.com/articles/layout-guide.html>

5. Making it our own: usability upgrades - shinythemes::themeSelector

Client-side code

```
ui <- fluidPage(  
  shinythemes::themeSelector(),  
  ... # whatever other front-end  
  code you're including  
)
```

th regression!



theme selector

37

bioinformatics.ca

`shinythemes::themeSelector(),`

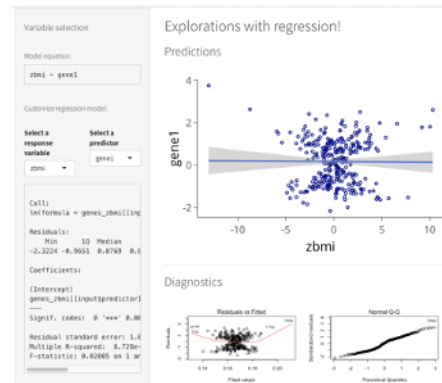
Can be included anywhere in your **ui** object, to allow you to play around with different themes before you commit to one that suits you

No server-side component is required to make this work! Simple include the shinythemes library at the top of your code with **library(shinythemes)**

5. Making it our own: usability upgrades - shinythemes::shinytheme

Client-side code

```
ui <- fluidPage(  
  theme =  
  shinythemes::shinytheme("cosmo"),  
  ... # whatever other front-end  
  code you're including  
)
```



38

bioinformatics.ca

Once you have chosen a suitable theme, specify it through the **theme = shinytheme("cosmo")** command, replacing "cosmo" with the name of the theme you prefer.

No server-side component or additional files are required! If you wanted to write your own HTML or CSS however, to augment this theme to your liking or even to write a theme from scratch, this file would live in the **www** directory we created at the start of the module

We are on an **input\$anything** Break

compute | calcul
canada | canada

Workshop Sponsors:



Canadian Centre for
Computational
Genomics



39

bioinformatics.ca